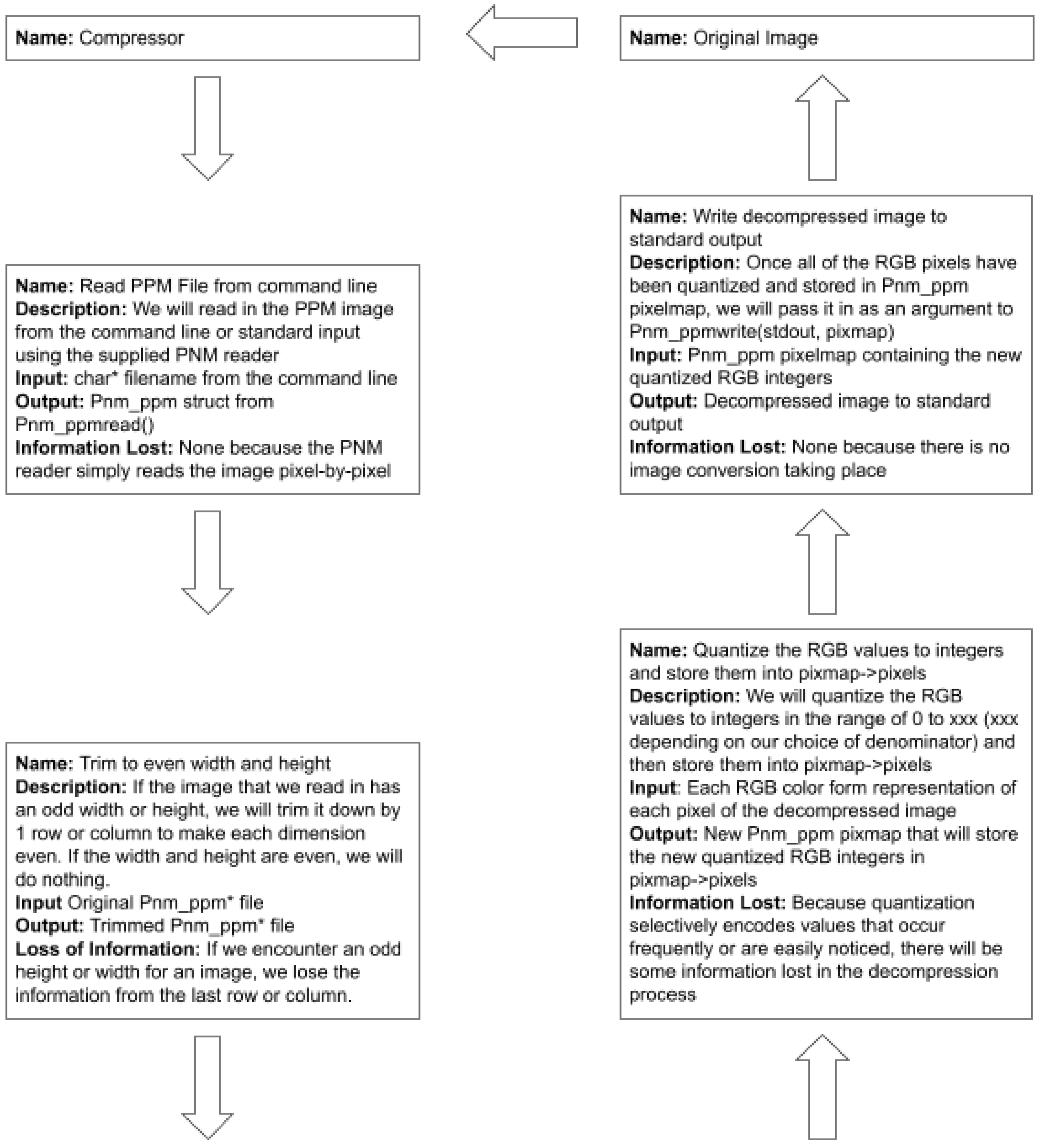
Brandon Dionisio (bdioni01) and Jordan Pauzie (jpauzi01)

Arith Box Flow Chart



Name: Compressor

Name: Read PPM File from command line
Description: We will read in the PPM image from the command line or standard input using the supplied PNM reader
Input: char* filename from the command line
Output: Pnm_ppm struct from Pnm_ppmread()
Information Lost: None because the PNM reader simply reads the image pixel-by-pixel

Name: Trim to even width and height
Description: If the image that we read in has an odd width or height, we will trim it down by 1 row or column to make each dimension even. If the width and height are even, we will do nothing.
Input Original Pnm_ppm* file
Output: Trimmed Pnm_ppm* file
Loss of Information: If we encounter an odd height or width for an image, we lose the information from the last row or column.

Name: Original Image

Name: Write decompressed image to standard output
Description: Once all of the RGB pixels have been quantized and stored in Pnm_ppm pixelmap, we will pass it in as an argument to Pnm_ppmwrite(stdout, pixmap)
Input: Pnm_ppm pixelmap containing the new quantized RGB integers
Output: Decompressed image to standard output
Information Lost: None because there is no image conversion taking place

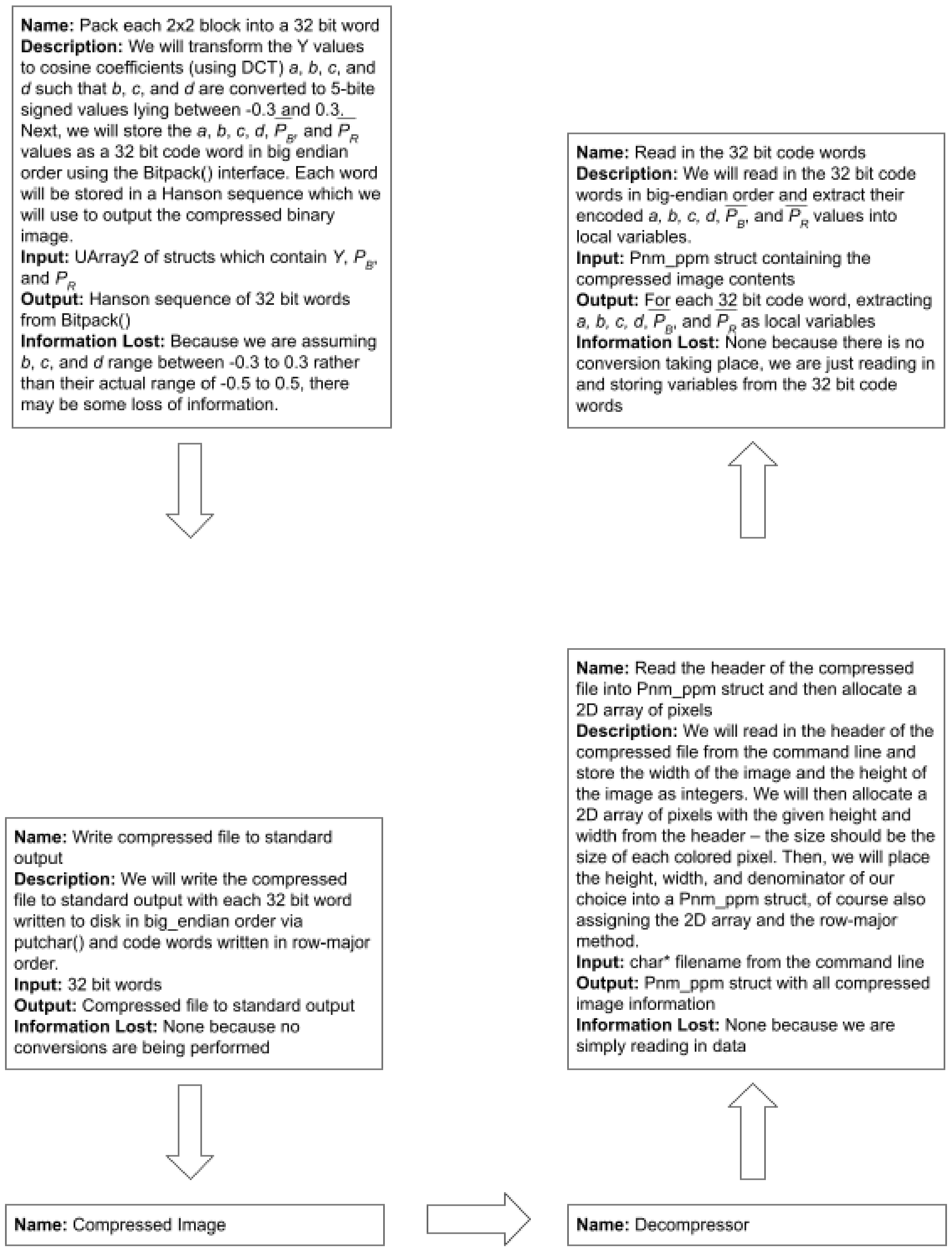
Name: Quantize the RGB values to integers and store them into pixmap->pixels
Description: We will quantize the RGB values to integers in the range of 0 to xxx (xxx depending on our choice of denominator) and then store them into pixmap->pixels
Input: Each RGB color form representation of each pixel of the decompressed image
Output: New Pnm_ppm pixmap that will store the new quantized RGB integers in pixmap->pixels
Information Lost: Because quantization selectively encodes values that occur frequently or are easily noticed, there will be some information lost in the decompression process

Name: Convert the pixels from RGB representation to floating point representation and then into component-video color space
Description: We will change the pixels into floating point representation and transform them into component video color space using the equations at the top of page 7 of the spec. In order to do this, we must create a new UArray2 and translate each of the old Pnm_rgb structs to new structs containing Y, P_B, and P_R in the new UArray2.
Input Old Pnm_ppm
Output: New UArray2 of structs which contain Y, P_B and P_R.
Information Lost: None because the RGB values should have one-to-one equivalents in component video color space

Name: Get chroma averages
Description: For each 2x2 block in row-major order, we will get the average $\overline{P_B}$ and $\overline{P_R}$ and store these values in a local variables. Then, we will store the averages as four-bit values using the provided Arith40_index_of_chroma function.
Input: UArray2 of structs which contain Y, P_B, and P_R
Output: Four bit representations of $\overline{P_B}$ and $\overline{P_R}$
Information Lost: Since we are taking the average of the four chroma values in the 2x2 block as one value, we are losing information on the variance of chroma values.

Name: Transform pixels from component-video representation to RGB color representation
Description: We will change the pixels in the 2x2 block from component-video representation to RGB color using the equations at the top of page 7 of the spec.
Input: Component-video representation ($\overline{P_B}$ and $\overline{P_R}$ chroma codes and Y_1, Y_2, Y_3, and Y_4) local variables for each pixel of the compressed image.
Output: RGB color form local variables for each pixel of the compressed image
Information Lost: None because the component-video representation should have one-to-one equivalents in RGB color

Name: Convert values into component-video representation
Description: We will convert the four-bit chroma codes to $\overline{P_B}$ and $\overline{P_R}$ using the provided function Arith40_chroma_of_index. Then compute Y_1, Y_2, Y_3, Y_4 from *a*, *b*, *c*, *d* using the inverse of the discrete cosine transformation.
Input: Local variables containing the encodings of *a, b, c, d*, $\overline{P_B}$, and $\overline{P_R}$ for each pixel of the compressed image.
Output: New $\overline{P_B}$ and $\overline{P_R}$ chroma codes and Y_1, Y_2, Y_3, and Y_4 local variables for each 2x2 block of the decompressed image.
Information Lost: None because we are simply using the function and the inverse of the DCT to transform the information in the encodings.



Name: Pack each 2x2 block into a 32 bit word
Description: We will transform the Y values to cosine coefficients (using DCT) *a*, *b*, *c*, and *d* such that *b*, *c*, and *d* are converted to 5-bite signed values lying between -0.3 and 0.3. Next, we will store the *a*, *b*, *c*, *d*, P_B, and $\overline{P_R}$ values as a 32 bit code word in big endian order using the Bitpack() interface. Each word will be stored in a Hanson sequence which we will use to output the compressed binary image.
Input: UArray2 of structs which contain *Y*, P_B, and P_R
Output: Hanson sequence of 32 bit words from Bitpack()
Information Lost: Because we are assuming *b*, *c*, and *d* range between -0.3 to 0.3 rather than their actual range of -0.5 to 0.5, there may be some loss of information.

Name: Write compressed file to standard output
Description: We will write the compressed file to standard output with each 32 bit word written to disk in big_endian order via putchar() and code words written in row-major order.
Input: 32 bit words
Output: Compressed file to standard output
Information Lost: None because no conversions are being performed

Name: Compressed Image

Name: Decompressor

Name: Read the header of the compressed file into Pnm_ppm struct and then allocate a 2D array of pixels
Description: We will read in the header of the compressed file from the command line and store the width of the image and the height of the image as integers. We will then allocate a 2D array of pixels with the given height and width from the header – the size should be the size of each colored pixel. Then, we will place the height, width, and denominator of our choice into a Pnm_ppm struct, of course also assigning the 2D array and the row-major method.
Input: char* filename from the command line
Output: Pnm_ppm struct with all compressed image information
Information Lost: None because we are simply reading in data

Name: Read in the 32 bit code words
Description: We will read in the 32 bit code words in big-endian order and extract their encoded *a*, *b*, *c*, *d*, $\overline{P_B}$, and $\overline{P_R}$ values into local variables.
Input: Pnm_ppm struct containing the compressed image contents
Output: For each 32 bit code word, extracting *a*, *b*, *c*, *d*, $\overline{P_B}$, and $\overline{P_R}$ as local variables
Information Lost: None because there is no conversion taking place, we are just reading in and storing variables from the 32 bit code words

Implementation Plan

In creating our program, we plan to incrementally implement and test each step starting with the original image. That is, we will first start by testing the functionality of the first compression step and last decompression step, then with the second compression step and second to last decompression step, and so on. Finally, once we verify that we can compress and decompress a given image seamlessly, we will implement the command line arguments to perform each action individually.

First compression step & last decompression step:

1. Read PPM File from command line
    a. We will test if the file was read successfully by asserting if Pnm_ppm != NULL, effectively throwing a CRE if it is NULL.
    b. Check to see if all the Pnm_ppm struct attributes are as expected.
2. Write decompressed image to standard output
    a. Pass in several ppm image files and see if we can store inside the Pnm_ppm and then print out again to display.
    b. Attempt to pass in images that are badly formatted to see if the program handles them correctly.
    c. Attempt to pass in images that don't exist to see if the program returns with EXIT_FAILURE as intended.

Second compression step:

3. Trim to even width and height
    a. Run the program with a ppm image file that has an even width and height. Test with a print statement to see if the image does not get trimmed.
    b. Run the program with a ppm image file that has an odd width and even height. Test with a print statement to see if the image width gets trimmed by 1.
    c. Run the program with a ppm image file that has an even width and odd height. Test with a print statement to see if the image height gets trimmed by 1.
    d. Run the program with a ppm image file that has an odd width and height. Test with a print statement to see if both the image height and width gets trimmed by 1.
    e. Run the program with an image with a width or height of 1. Test to see if we exit the program properly (since we can't trim to 0).

Third compression step & fifth/sixth decompression steps:

4. Convert the pixels from RGB representation to floating point representation and then into component-video color space
    a. Print out each conversion to verify the values are expected
5. Transform pixels from component-video color space to RGB color representation
    a. Assert that the original RGB color floating point representations == the transformed RGB color representations that we reverted
    b. Verify that all the values are between 0 and 1.
6. Quantize the RGB values to integers and store them into pixmap->pixels
    a. Verify that the values are less than the denominator and greater than 0.
    b. Run the program with various ppm image files to see if we can output the original image using the compression and decompression steps implemented so far

Fourth compression step & fourth decompression step:

7. Get chroma averages
    a. Verify the functionality of this step on its own using dummy values. Pass in a 2x2 block of which we know the P_B and P_R values and verify that the averages are correct.
    b. Check to see that the outputted $\overline{P_B}$ and $\overline{P_R}$ are 4 bits each with a size function.
8. Convert values into component-video representation
    a. First, we will test this part's functionality on its own and then with the Bitmap interface.
    b. First, assert that each of the $\overline{P_B}$, $\overline{P_R}$, a, b, c, and d values are their correct bit sizes in their local variables.
    c. Print out the converted $\overline{P_B}$ and $\overline{P_R}$ to verify their correctness based on an encoding which we know the correct values.
    d. Then, print out the converted Y_1, Y_2, Y_3, and Y_4 values after the inverse of the DCT to verify their correctness based on an encoding which we know the correct values.

Fifth compression step & third decompression step:

9. Pack each 2x2 block into a 32-bit word
    a. Verify the functionality of our Bitmap interface by testing out the value conversions on their own. Pass in a 2x2 block of which we know the output word and verify that it is correct.
    b. Check to see that the outputted b, c, and d are 5 bits each with a size function. Check to see that the outputted a is 9 bits with a size function.
    c. Test that we are able to read the word in big endian order.
    d. We will assert that a, b, c, and d are lesser than or equal to the absolute value of 0.3 and also assert that the 32-bit word created by Bitpack() != NULL. If these assertions fail, we will throw a CRE.
    e. If all the above tests pass, pass in a ppm image file to the program and verify that the outputted 32 bit word is expected.
10. Read in the 32 bit code words
    a. First, test this functionality on its own:
    b. With a 32 bit word, test to see if we can unpack the $\overline{P_B}$, $\overline{P_R}$, a, b, c, and d values by printing out the local variables.
    c. Assert that each of the local variable values have the correct bit size – throw a CRE if not.
    d. Then, with images:
    e. Run the program with various ppm image files to see if we can output the original image using the compression and decompression steps implemented so far.

- *From here, we can stop testing the program as a cycle of compression and decompression and test each image transformation individually.*

Sixth compression step & first/second decompression steps:

11. Write compressed file to standard output
    a. Assert that the size of our sequence holding the 32 bit words is the height * width / 4 of the trimmed original image.
    b. At this point, we should be able to handle compression through the command line, so we can run the program with various ppm image files and see if we can output the compressed images properly.
    c. Test to see if the compressed images have the correct dimensions.
    d. Test to see if the compression part of the program handles badly formatted images, extremely large images, and other edge cases.
12. Read in the header of compressed file into a 2D-array and Pnm ppm struct
    a. Assert that none of the width, height, or number of input items that we read in from the command line are NULL.
    b. Assert that the initialized UArray2 != NULL and the initialized Pnm_ppm struct != NULL.
    c. Verify with print statements that all the attributes of Pnm ppm struct are the values they are supposed to be.
    d. At this point, we should be able to handle decompression through the command line, so we can test that the entire decompression process functions with various ppm image files.
    e. Test to see if the decompression part of the program handles badly formatted images, extremely large images, and other edge cases.